NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE LOCAL MOTION PLANNING FOR AN AUTONOMOUS MOBILE ROBOT

by

Seok Jun Yun

September 1995

Thesis Advisor: Yutaka Kanayama

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 1

19960220 036

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1995	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
THE LOCAL MOTION PLANNING FOR AN AUTONOMOUS MOBILE ROBOT

5. FUNDING NUMBERS

6. AUTHOR(S)
Yun, Seok Jun

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The major problem addressed by this research is how to develop a motion control algorithm for local motion planning of an autonomous mobile robot.

The approach taken was to clearly define the robot's motion descriptions and to design a high-level, machine independent robot control language called MML (Model-based Mobile robot Language).

The results are that the robot can implement smooth rotation, symmetric, and initial motion tracking on an given environment. Based on the motion control algorithm developed in this thesis, the robot with rigid body can be applicable in both local and global motion planning.

The experimental results were successful. The algorithms were implemented first on a simulator, then on the autonomous mobile robot *Yamabico-11*. Given an initial and final configuration, the vehicle demonstrated the capability to safely achieve its goal.

14. SUBJECT TERMS
Robotics, autonomous mobile robots, Motion Control

15. NUMBER OF PAGES
74

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

THE LOCAL MOTION PLANNING FOR
AN AUTONOMOUS MOBILE ROBOT

Seok Jun Yun
Captain, ROKArmy
B.S., Korea Military Academy, 1988

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
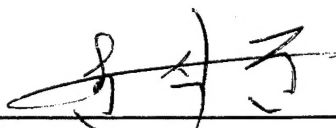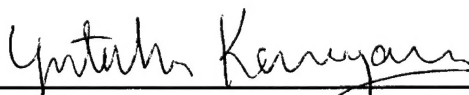
from the

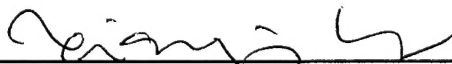**NAVAL POSTGRADUATE SCHOOL**

September 1995

Author: _____

Seok Jun Yun

Approved by: _____

Yutaka Kanayama, Thesis Advisor

_____

Xiaoping Yun, Second Reader

_____

Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The major problem addressed by this research is how to develop a motion control algorithm for local motion planning of an autonomous mobile robot.

The approach taken was to clearly define the robot's motion descriptions and to design a high-level, machine independent robot control language called MML (Model-based Mobile robot Language).

The results are that the robot can implement smooth rotation, symmetric, and initial motion tracking on an given environment. Based on the motion control algorithm developed in this thesis, the robot with rigid body can be applicable in both local and global motion planning.

The experimental results were successful. The algorithms were implemented first on a simulator, then on the autonomous mobile robot *Yamabico-11*. Given an initial and final configuration, the vehicle demonstrated the capability to safely achieve its goal.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

One of the ultimate goals in robotics is to develop an autonomous robot - being capable of successfully completing a task given what to do and how to do it. Increasingly capable autonomous robot platforms are being developed to handle a myriad of hazardous duty assignments. While manufacturing dominates the area of robotic applications, useful advances have been made in the areas of waste management, space exploration, undersea work, assistance for the disabled and medical surgery [Lato]. Several government-sponsored efforts are underway for building systems for military applications such as fighting fires, handling ammunition, transporting material, conducting underwater search and inspection operations, and other dangerous tasks now performed by humans [Nava].

Many of the above tasks require motion of the robot in order to carry out any task. This problem is known as the motion planning problem. We divide the motion planning problem into two sub-problems: the global and local motion planning. The robot motion can be described as follows: a path of robot to carry out a task while moving from one configuration (p, θ, k) to another. In motion planning, not only is its position important, but the orientation and curvature of the robot are important as it follows the path.

The purpose of the research is to investigate fundamental theories for navigation to construct autonomous mobile robot for military and industrial applications. In this thesis, we will discuss several local motion planning problems inevitable for the robot to carry out any task. They are divided as follows: smooth rotation, symmetric, and initial motion planning.

II. BACK GROUND

The focus of this thesis is on planning the motion of an autonomous mobile robot. The algorithms for motion planning to be taken by real robot are developed. Many simulations are done to verify the algorithms.

A high-level motion specification language, Mobile-based Mobile-robot Language (MML-11), suitable to describe the solutions to the motion planning problem is used as the implementation and verification of the algorithms.

The world space W for the motion planning problem in this thesis is a two dimensional plane $\mathfrak{R}^2$ on which a global Cartesian coordinate system is defined. The obstacles are closed subsets of W. The free space, $FREE(W)$, is the complement of the union of all obstacles in the world space W.

The vehicle in the thesis refers to a rigid body robot which has fixed shape - during the research we will use *Yamabico-11* for experiments although the algorithms will be suitable for robots with any variable shape.

A configuration q in this thesis defined as a combination of position, orientation, and curvature, (p, θ, k), where p indicates the position (x, y) in the global Cartesian coordinate system, θ is the orientation related to the x-axis of the global coordinate system [Loza], and k is the specified curvature. The configuration defined in this thesis is normally used to describe the robot's instantaneous status, either it is stationary or moving. This configuration is specially useful to specify a path. For instance, if we use $q = ((x, y), \theta, k)$ to specify a line, this line passes through the point at (x, y) and with orientation θ. When the curvature element $k = 0$, it is specifying a straight line, otherwise it's a circle.

3

The motion of a vehicle is subject to nonholonomic and kinematic constraints, that is, the vehicle is able to perform both forward and backward motion but not sideways motion. The robot's orientation and curvature at the path are continuous.

III. PROBLEM STATEMENTS

The problem investigated in this thesis is intelligent behavior of autonomous mobile robots with emphasis on local motion planning and how to precisely and stably control the motions of the autonomous mobile robot *Yamabico-11*. This problem is subdivided into the following problems:

1. What procedures should an autonomous mobile robot follow in order to rotate smoothly in a given environment?

2. What procedures should an autonomous mobile robot follow to get to the configuration of not being on the center of the path?

3. What procedures should an autonomous mobile robot follow to avoid the collision of the walls and to get to the goal configuration?

IV. SMOOTH ROTATION

A.   MOTION DESCRIPTION

We consider a situation where the vehicle at an arbitrary configuration $q_s = (p_s, \theta_s, k_s)$ is supposed to rotate its orientation from θ_s to an arbitrary θ_g without changing its position p_s. (Figure 1)
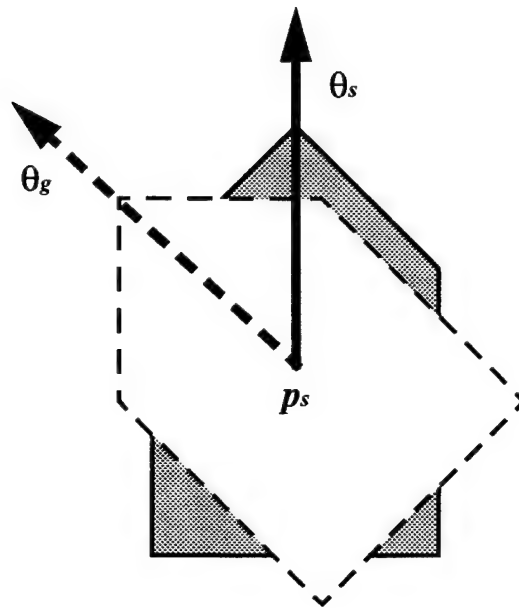
Figure 1: Rotating of orientation without changing its position

The motion of this vehicle can be described as follows:

Start rotating the orientation towards the desired direction.

Check the vehicle heading.

Stop rotating when its heading equals to the goal orientation.

As we can see, the motion does not involve any translation of the vehicle. So it is not nec-

essary to use the steering function. Instead, the motion is based on relation between difference of angles and rotational speed of the vehicle. We need an algorithm which controls the rotational speed of the vehicle depending on the vehicle's rotational angle.

B.   MOTION PLANNING

1.   Algorithm For Smooth Rotation

The algorithm as Eq 4.1 is a tool for a mobile robot vehicle performing smooth rotation.

$$\frac{dw}{dt} = -2kw - k^2(\theta_s - \theta_g) \tag{Eq 4.1}$$

where $\frac{dw}{dt}$: Vehicle's rotational acceleration,

w: Vehicle's rotational speed,

θ_s: Vehicle's initial orientation,

θ_g: Vehicle's goal orientation,

k: Positive constant

As we can see in Eq 4.1, the rotational speed is the only control variable of the algorithm. While rotating, the algorithm returns rotational acceleration depending on the current speed at each time, and hence, the speed of the vehicle also.

2.   Motion Planning With Algorithm

In Eq 4.1, the positive constant k is an important factor of the algorithm. The bigger k we use, the rotational time is shorter, and the smaller k, it takes more time to rotate. Figure 2 shows the relation between speed and time for the vehicle to rotate π degrees with various k.

8
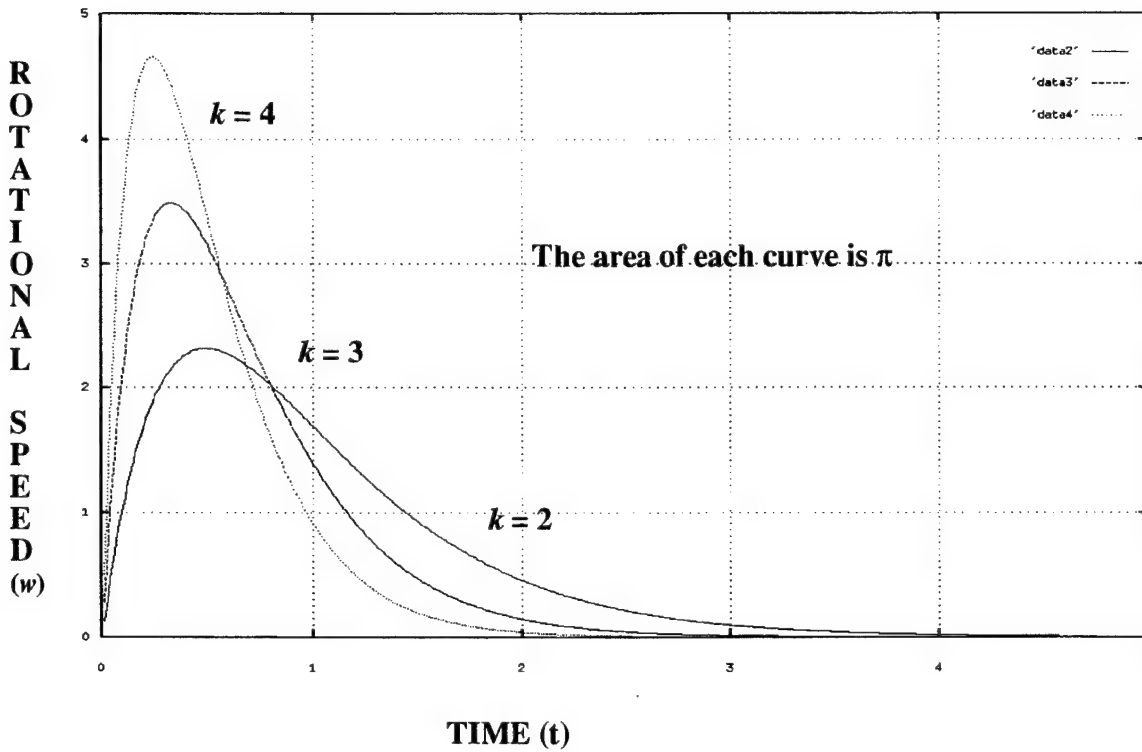
Figure 2: The graph for smooth rotation algorithm

As shown in the figure 2, using bigger k, for example, $k = 4$ in the graph, gives the vehicle higher speed and shorter time to complete its mission But in some cases, it can be dangerous for the vehicle if the speed exceeds the vehicle's speed limit. There can be no such high speed in using smaller k, but it has two defects: one being too much time to do its job, and the other being the lower speed duration time is too long.

C.   EXPERIMANTAL RESULT

Considering rotational time, we used $k = 4$ curve, but for the safety of the vehicle, we limit the speed with $w = 3$. Figure 3 shows the result.
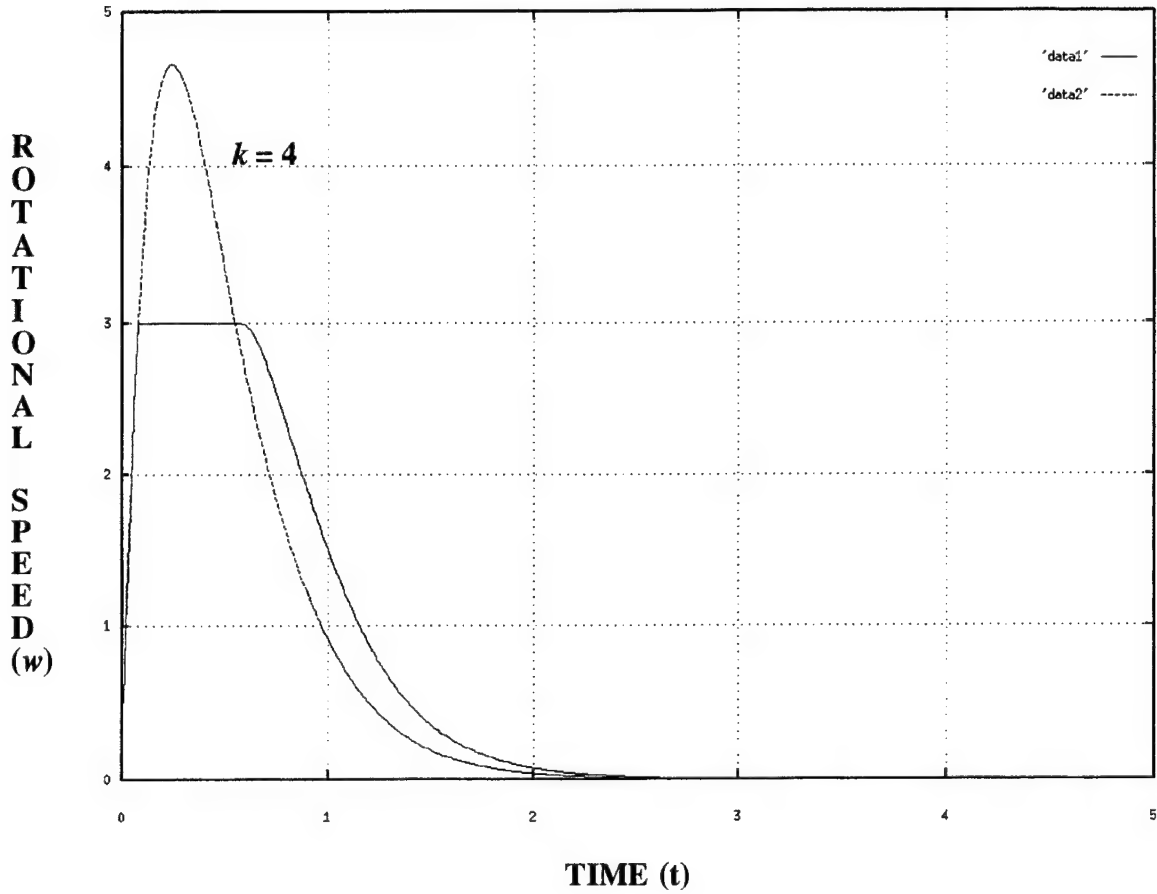
Figure 3: The graph for redefined algorithm

As shown by the result, applying the speed limit makes the algorithm more useful.

V. SYMMETRIC MOTION PLANNING

A.   MOTION DESCRIPTION

We consider a situation where the vehicle is supposed to track a path, and to get to the goal position which may not be on the center of the path. A vehicle is represented by a point $p_s = (x_s, y_s)$, its orientation α_s, and curvature k_s, and the goal position is represented by $p_g = (x_g, y_g)$, and its orientation α_g, and its curvature k_g. Figure 4 shows the situation.
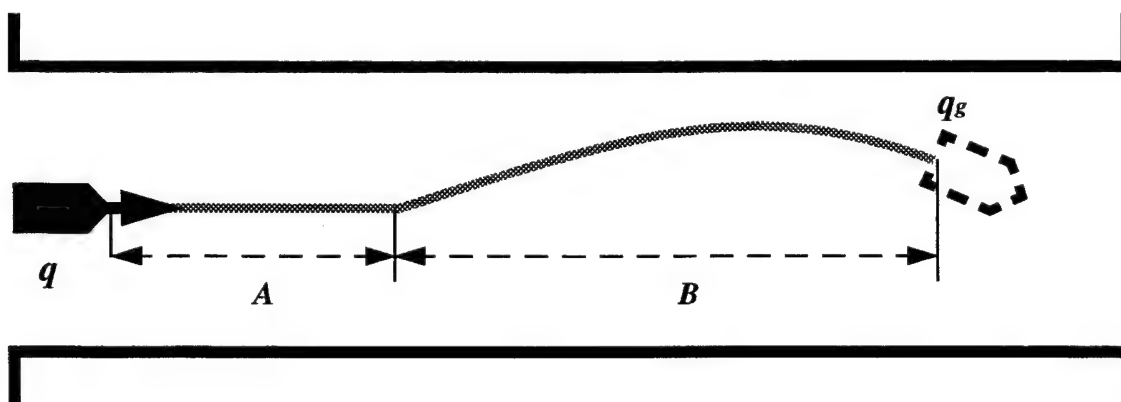
Figure 4: Symmetric motion

The motion of this vehicle can be divided into two parts; one being the motion on the path A, and the other being the motion on the curve B. The first part of the motion is easily done by the steering function, but the motion on the curve B is difficult to make it using the transformation. For the safety of vehicle, it is important to keep moving on the center of path as long as possible.

B.   MOTION PLANNING WITH STEERING FUNCTION

1.   Line Tracking

Let's assume a directed straight line L represented by a point (a, b) on it, and its orienta-

tion α. A vehicle away from the line is supposed to track the line. The vehicle is represented by *p* = *(x, y)*, and its orientation θ. (Figure 5)
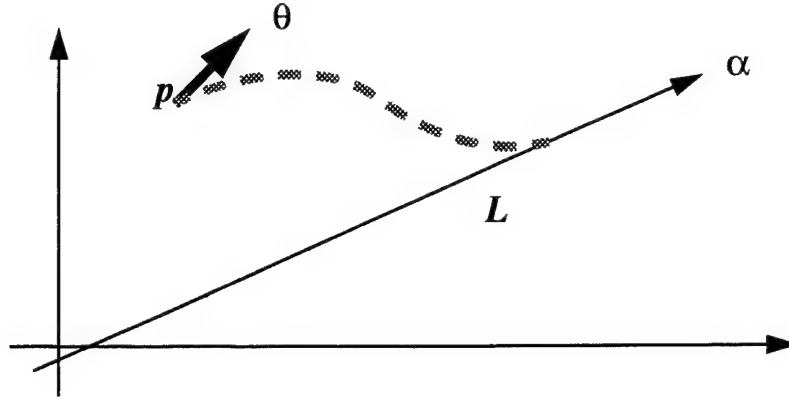
Figure 5: Line tracking

The steering function for the vehicle supposed to track the line *L* is [Kana]

$$\frac{dk}{ds} = -(ak + b(\theta - \alpha) + c(-(x-a)\sin\alpha + (y-b)\cos\alpha))$$ (Eq 5.1)

Using this steering function, a vehicle tracks a dotted line as shown in Fig. 5.2. How the steering function accomplishes a line tracking is fully described in [Kana], [Macp], and [Kova].

2.   Symmetric Motion

As seen in Section A in this Chapter, the motion *B* is a reverse action of line tracking. Let's assume a virtual vehicle which tracks a line from goal position. There will be a path kept by the virtual vehicle. With the path in the reverse direction, the real vehicle can move toward the goal position.

a.   *Virtual Vehicle Simulation*

We let

Start Configuration: $q_s = ((x_s, y_s), q_s, k_s)$,

Goal Configuration: $q_g = ((x_g, y_g), q_g, k_g)$

Let's assume there is a virtual vehicle at the goal position with reverse direction. The virtual vehicle is supposed to track a directed straight line L represented by the start configuration q_s.
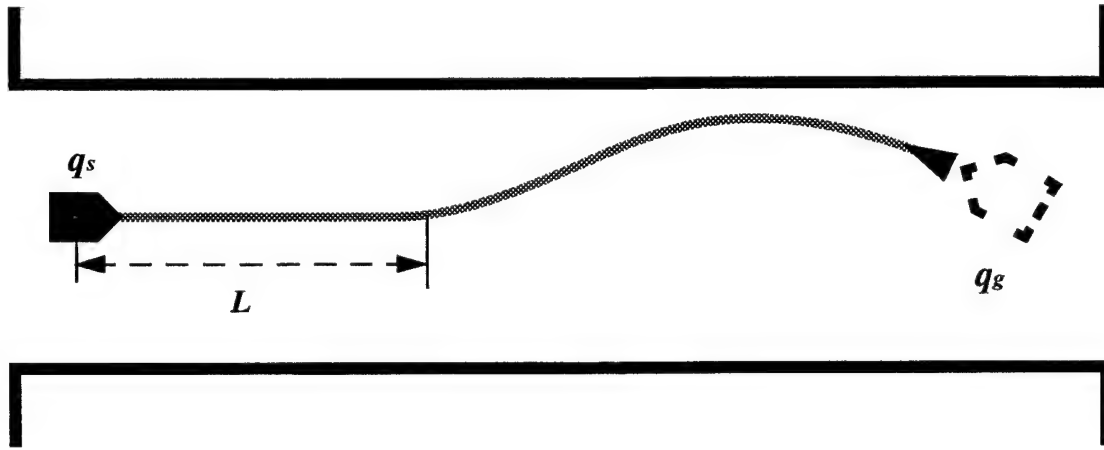
Figure 6: Virtual vehicle tracking

We can get

$k_d = -k_s$,

$\theta_d = \theta_s + \pi$,

$\Delta d = -(x - x_s)\sin\theta_d + (y - y_s)\cos\theta_d$,

because the virtual vehicle uses the q_s as its goal configuration.

The virtual vehicle steering function is

$$\frac{dk}{ds} = -(ak + b(\theta - \theta_d) + c(-(x - x_s)\sin\theta_d + (y - y_s)\cos\theta_d)) \qquad \text{(Eq 5.2)}$$

Using this steering function, the virtual vehicle tracks the line L, and finally gets to

the start configuration. Configurations generated during the virtual vehicle move are stored in array to be used by real vehicle later. When storing configurations, θ is changed to $\theta + \pi$, and kappa is changed its sign(+/-).

b. Real Vehicle Moving while Simulation

While the virtual vehicle simulates a line tracking, the real vehicle moves forward also. There will be some point between start configuration q_s and goal configuration q_g where the real vehicle meets the virtual one. The point has an important role in symmetric motion planning, because the virtual vehicle finishes its line tracking simulation, and the real vehicle starts tracking a simulated path at the point. We need to know how to get the point.
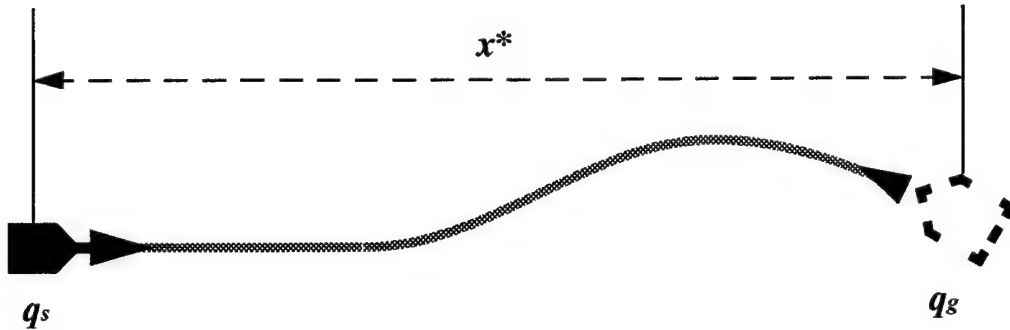
Figure 7: Real vehicle moving

As shown in Figure 7, we can get the distance between the real vehicle and the virtual one by computing the local coordinate x* - detailed explanation for the local coordinate will be discussed in later subsection. When the distance equals to 0, it is the time the real vehicle starts tracking a simulated path generated by virtual vehicle.

c.   *Local Coordinate*

We take simulated configurations by the virtual vehicle sparser than those gener-ated by the real vehicle. In addition to the data, the real vehicle uses the steering function locally between the configurations.

Let's assume

Vehicle configuration: $q = ((x, y), \theta, k)$,

Target configuration: $q_c = ((x_c, y_c), \theta_c, k_c)$

To use the steering function, we need to compute a local coordinate $q^* = ((x^*, y^*), \theta^*, k^*)$ between vehicle and target configuration. (Figure 8)
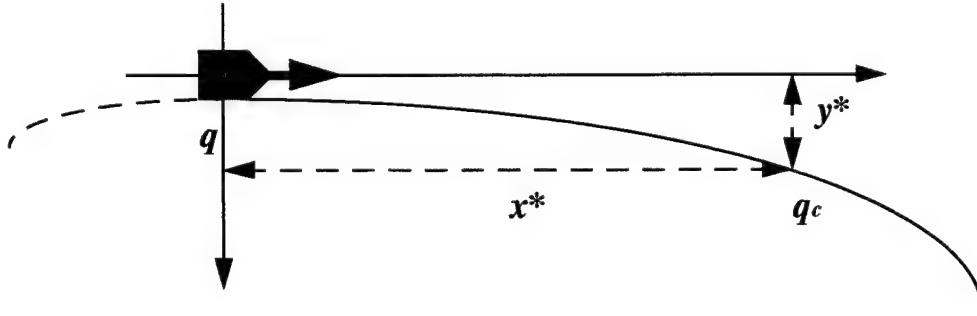
Figure 8: Local coordinate

The step how to compute local coordinate q^* is as follows:

$q_c = q \circ q^*$,

$q' \circ q_c = (q' \circ q) \circ q^*$

As a result, we get

$q^* = q' \circ q_c$ (Eq 5.3)

With the Eq 5.3, we can compute local coordinates as follows [Kana95]

$$x^* = (x - x_c)\cos\theta + (y - y_c)\sin\theta \qquad\qquad (Eq\ 5.4)$$

$$y^* = (x - x_c)\sin\theta + (y - y_c)\cos\theta \qquad\qquad (Eq\ 5.5)$$

d.   *Real Vehicle Tracking*

As shown in Fig. 5.4, the real vehicle uses a configuration of the simulated path one by one as its target. The step how to track the simulated path is as follows:

(1) Compute local coordinate between vehicle and target configuration

(2) Check x*

   (2.1) If x* <= 0

      (2.1.1) If the target is goal configuration

         (2.1.1.1) Stop vehicle

      (2.1.2) If the target is not goal configuration

         (2.1.2.1) Change the target with a new configuration of the simulated path

   (2.2) If x* > 0

      (2.3) Move the vehicle using steering function

      The real vehcle uses the steering function as follows:

$$\frac{dk}{ds} = -(a\Delta k + b\Delta\theta + c\Delta d) \qquad\qquad (Eq\ 5.6)$$

where $\Delta k = k - k_c,$

          $\Delta\theta = \theta - \theta_c,$

          $\Delta d = y^*$

16

C. EXPERIMENTAL RESULT

Figure 9 shows the result in which the real vehicle uses the same smoothness as the virtual vehicle does. The vehicle's initial configuration is (0, 0, 0). The vehicle's goal position is (400, 60) and its orientation is 0 degrees. As a result, the real vehicle gets to the configuration (400, 63, 0).
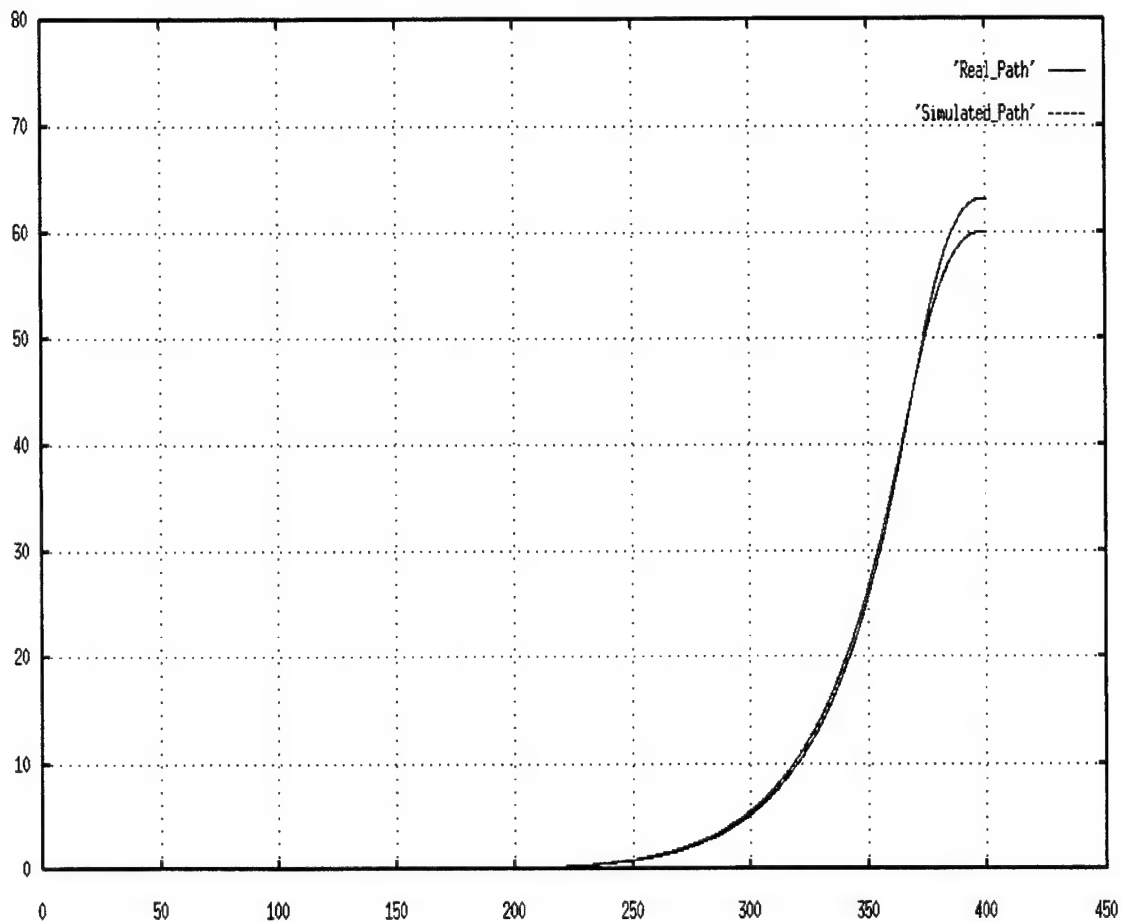
Figure 9: The result with the same σ as the virtual vehicle does

Figure 10 shows the result in which the real vehicle uses one half smoothness of the virtual vehicle. The vehicle's initial configuration is (0, 0, 0). The vehicle's goal position is (400, 60) and its orientation is 0 degrees. As a result, the real vehicle gets to the configuration (400, 60.5, 0).
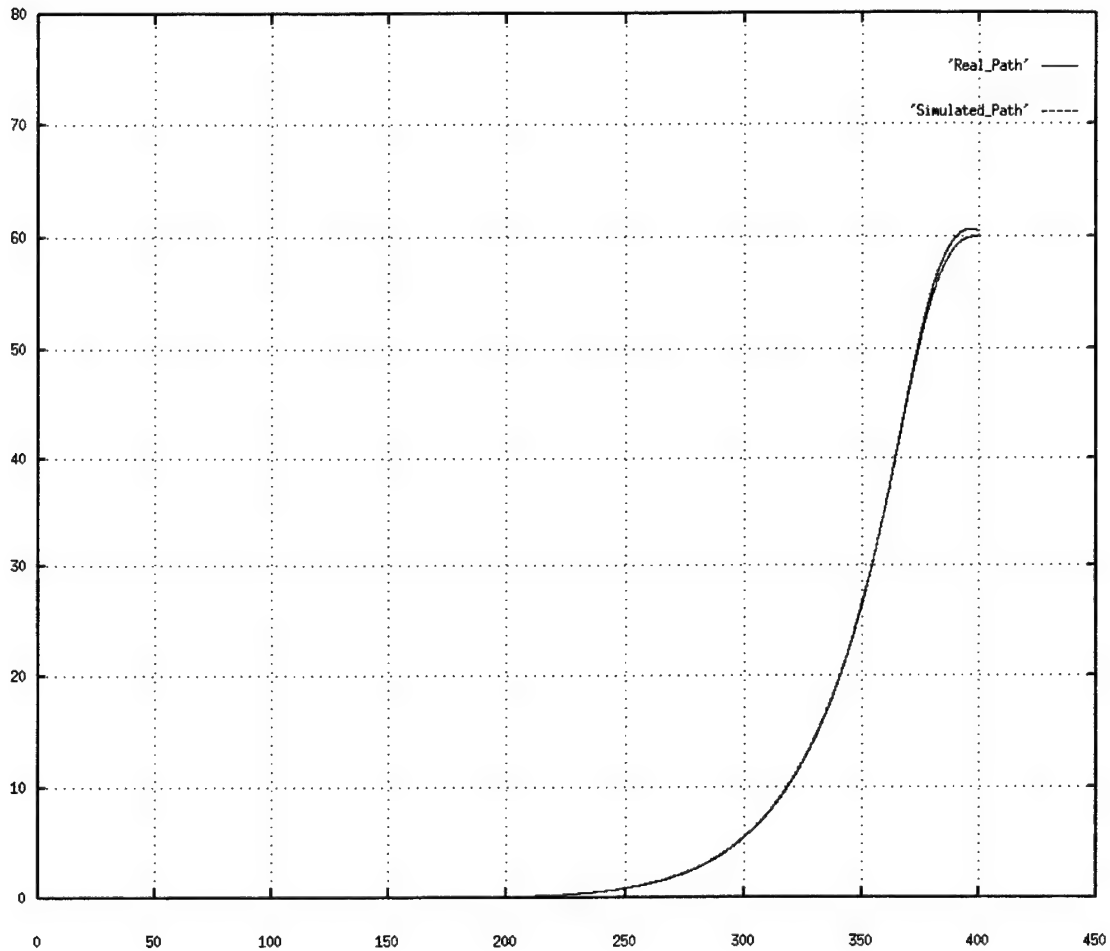
Figure 10: The result with a half of virtual vehicle σ

Figure 11 shows the result in which the real vehicle uses one quarter smoothness of the virtual vehicle. The vehicle's initial configuration is (0, 0, 0). The vehicle's goal position is (400, 60) and its orientation is 0 degrees. As a result, the real vehicle successfully gets to the configuration (400, 60, 0).
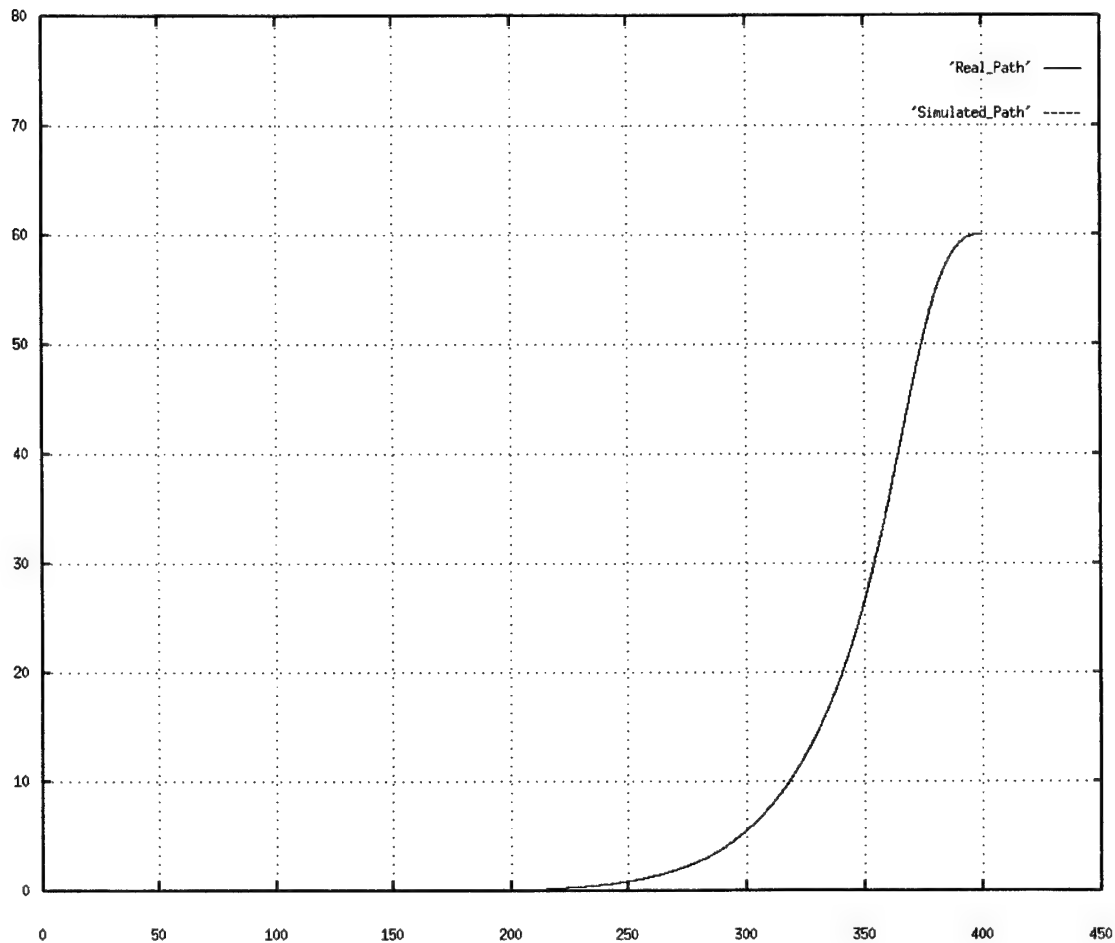
Figure 11: The result with a quarter of virtual vehicle σ

VI. INITIAL MOTION PLANNING

A. MOTION DESCRIPTION

We consider a situation where the vehicle at an arbitrary configuration $q_s = (p_s, \theta_s, k_s)$ is located on the world which is consisted of walls. There is one border which has a configuration called Exit configuration $q_e = (p_e, \theta_e, k_e)$. Figure 12 illustrates this situation.
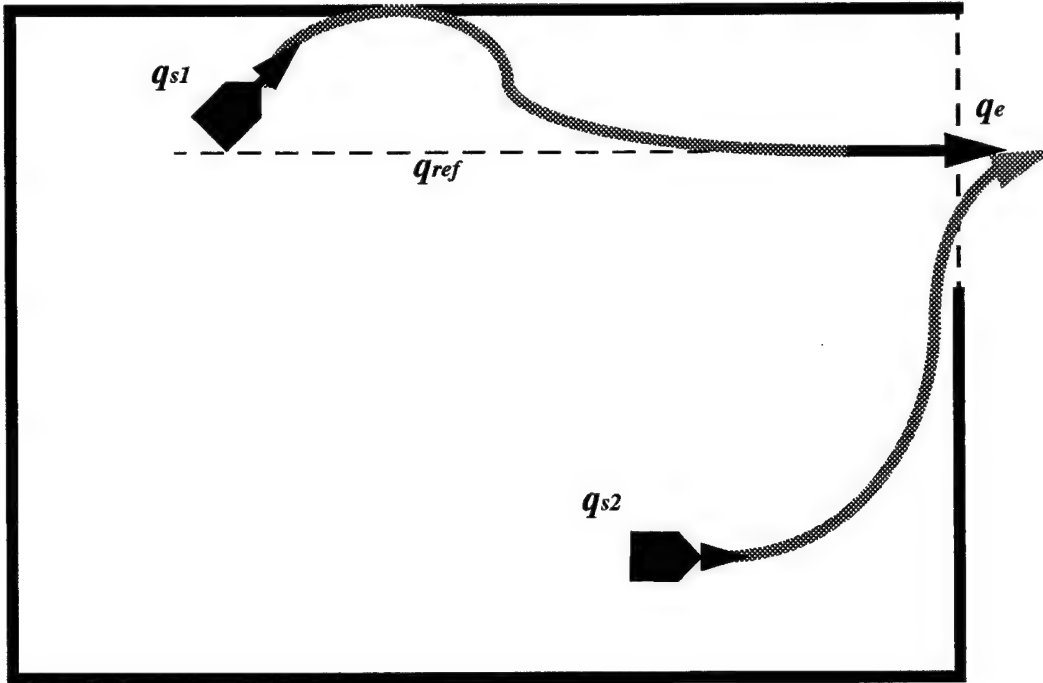
Figure 12: The case where simulated path collides

The world where vehicles are located is a part of k-regions [Kana95][Kova]. A vehicle located in the world has a mission to find a path which can converge to the exit configuration q_e without colliding the wall. The mission is a kind of line tracking, that is, the vehicle with a configuration q_s is supposed to track a straight line q_{ref} represented by p_e and its orientation θ_e using

steering function. As shown in the figure, since the vehicle at configuration q_{s1} is too close to the wall, the vehicle will collide the wall. For the safety of the vehicle, this behavior can not be accepted. The other vehicle at configuration q_{s2} is far away from the wall, but the vehicle fails to pass the exit configuration q_e. The exit configuration q_e is the start configuration of next region. It's important for the vehicle to get to the exact configuration. The mission can be divided into two parts: one being the mission to find a path to avoid colliding of walls, and the other being the mission to find a path to get to the exit configuration q_e.

B.   MOTION PLANNING WITH STEERING FUNCTION

The steering function (Eq 6.1) is used for a vehicle to perform smooth line tracking.

$$\frac{dk}{ds} = -(A\Delta k + B\Delta\theta + C\Delta d) \qquad \text{(Eq 6.1)}$$

In Eq 6.1, A, B, and C are positive constants which are related to the smoothness of vehicle's motion [Kana]. They are determined by the smoothness σ as follows:

$$k = 1 / \sigma \qquad \text{(Eq 6.2)}$$

$$A = 3 * k \qquad \text{(Eq 6.3)}$$

$$B = 3 * k * k \qquad \text{(Eq 6.4)}$$

$$C = 3 * k * k * k \qquad \text{(Eq 6.5)}$$

From the above equations, we know the smoothness σ determines the sharpness of the tracking trajectory. We can see that the smaller the value of smoothness is, the sharper the trajectory will be, and the larger the smoothness is, the longer the vehicle travels [Kana].

While we know tracking trajectory depends on smoothness, it is important to know that a lower limit of smoothness exists in using the steering function to track a line. The lower limit of smoothness is as follows [Chuang]:

$$\sigma = 0.095 * \Delta d_{init} \qquad \text{(Eq 6.6)}$$

22

In Eq 6.6, Δd_{init} is the closest distance from the initial configuration to the reference line. If a smaller smoothness σ than the lower limit is applied, the tracking trajectory never converge to the reference line.

In addition to the lower limit of smoothness, we have to consider there will be undesirable tracking trajectories to converge to the reference line with some arbitrary smoothness. Although they converge to the reference line, the trajectory travels backward. The minimum desirable smoothness for parallel line tracking is as follows [Chuang]:

$$\sigma = 0.18 * \Delta d_{init} \qquad\qquad (Eq\ 6.7)$$

1. Motion Planning to Avoid The Collision

A line tracking motion starts from a configuration $q_s = (p_s,\ \theta_s,\ k_s)$. The goal of the line tracking is the line called reference line q_{ref}, specified by a configuration $q_e = (p_e,\ \theta_e,\ k_e)$. Since the line is a straight line, it has a constant curvature $k_e = 0$. Figure 13 shows the case.
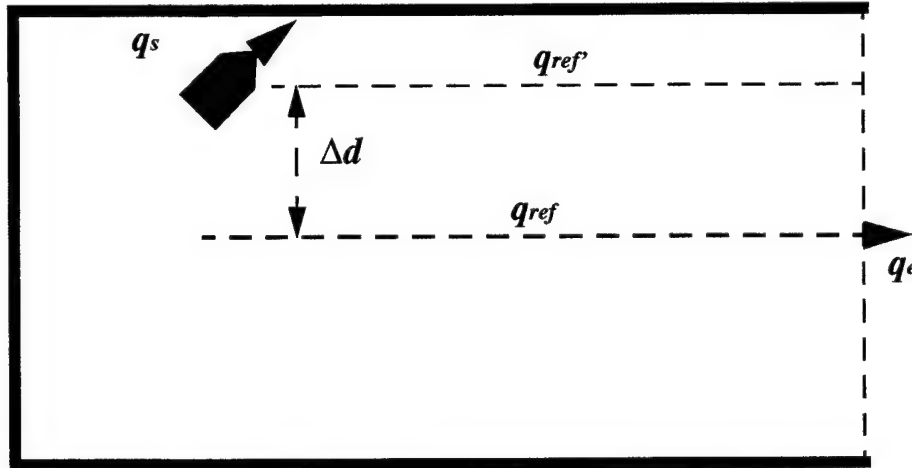
Figure 13: The case to avoid collision

The path of line tracking in the figure refers to a straight line q_{ref}. But we can make the Eq 6.1 simpler using q_{ref} as a reference line instead of using q_{ref}. We can use a bigger smoothness to converge to the reference line, because the Δd is 0.

$$\frac{dk}{ds} = -(A\Delta k + B\Delta\theta) \tag{Eq 6.8}$$

The step to avoid collision is as follows:

(1) Start line tracking simulation with default smoothness

(2) Check if path collides the wall

    - Checking collision will be discussed later in this chapter

    (2.1) If the path collides the wall

        (2.1.1) Change the smoothness to a smaller one

            (2.1.1.1) Check if the smoothness is smaller than $0.095 * \Delta d_{init}$

                (2.1.1.1.1) Quit the simulation

                (2.1.1.1.2) No path to avoid collision exists

        (2.1.2) Start line tracking simulation again with new smoothness

    (2.2) If there is no collision on the path

        (2.2.1) Check if the path is parallel to the exit configuration q_e

        (2.2.2) If the path is parallel to the exit configuration, stop the simulation.

(3) Return the smoothness σ

2. Motion Planning to Converge to The Reference Line

A line tracking motion starting from a configuration $q_s = (p_s, \theta_s, k_s)$ reached to a configuration $q_n = (p_n, \theta_n, k_n)$ which is parallel to the exit configuration q_e. From the configuration $q_n = (p_n, \theta_n, k_n)$, another line tracking motion starts to converge to the reference line q_{ref} specified by $q_e = (p_e, \theta_e, k_e)$. Figure 14 shows this case.
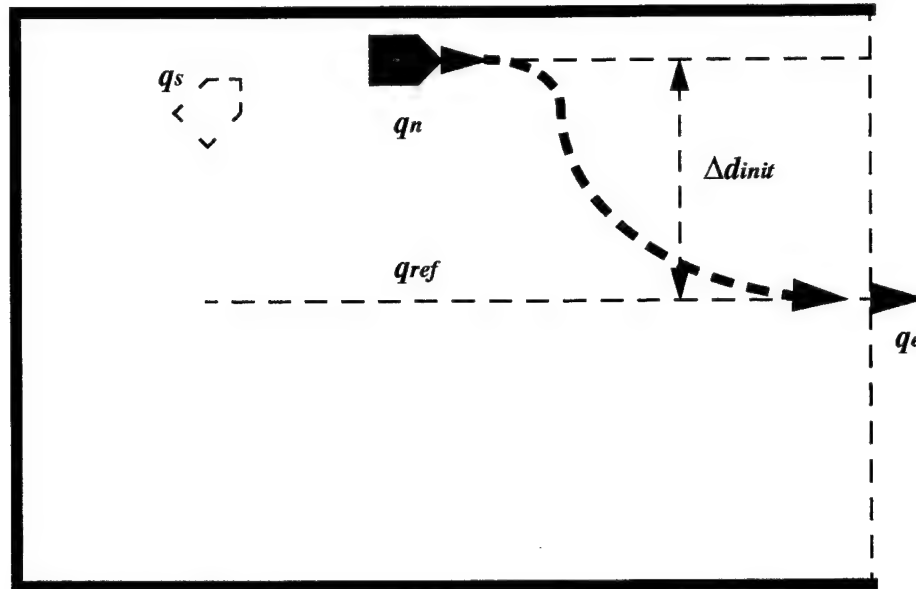
Figure 14: The case to converge to the reference line

The path of line tracking in the figure refers to a straight line q_{ref}. A line tracking starts with default smoothness σ again. The reason not using smoothness which we got in Section 1 is that the simulated path could be sharper to avoid the collision, that is, the smoothness would be smaller than $0.18 * \Delta d_{init}$. A line tracking path with the smoothness will be undesirable to converge to the reference line - the detailed explanation is discussed in Section 1 in this Chapter.

The step to converge to the reference line is as follows:

(1) Start line tracking simulation with default smoothness

(2) Check if x* equals to 0, or smaller than 0

    (1.1) Check if y* equals to 0

        (1.1.1) Return the smoothness σ

        (1.1.2) Quit the simulation

(1.2) If y* does not equal to 0

    (1.2.1) Change the smoothness to a smaller one

    (1.2.2) Check if the smoothness is smaller than $0.18 * \Delta d_{init}$

        (1.2.2.1) Quit the simulation

        (1.2.2.2) No path exists

(3) Start line tracking simulation again with new smoothness

3.   Motion Planning to Determine The Direction of Path

A line tracking motion starting from a configuration $q_s = (p_s, \theta_s, k_s)$ will get to the goal configuration $q_e = (p_e, \theta_e, k_e)$ on one desirable path. But actually, regarding to the heading of the vehicle, there can exist more desirable path. Figure 15 shows this case.
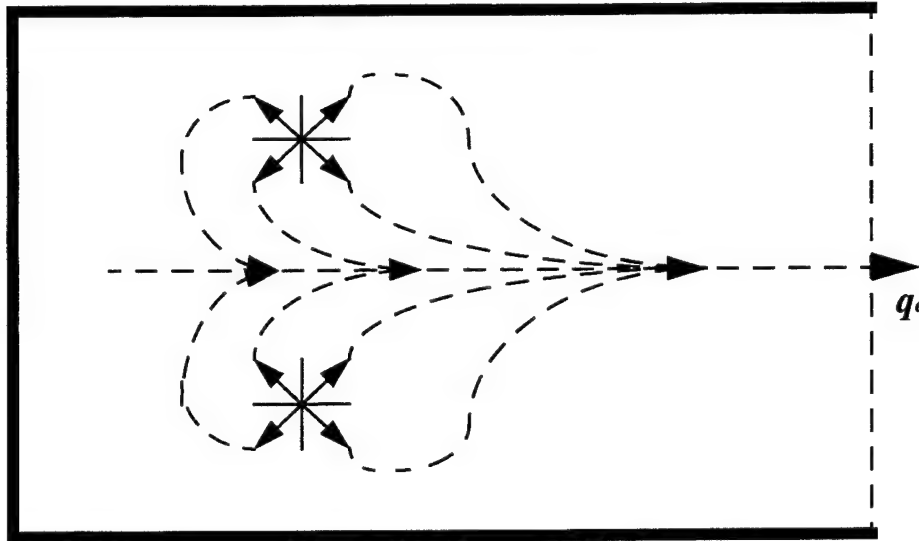
Figure 15: The case of various direction of paths

The direction of the path is related not only to the heading of the vehicle θ_s, but to the

orientation of the Exit configuration θ_e.

The step to determine the direction of the path is as follows:

(A) $|\Phi(\theta s - \theta e)| <= \pi / 2$

- Φ means normalization

   (1) If $((\theta s - \theta e) > \pi / 2)$ Then $\theta = \theta s - 2\pi$

   (2) Else if $((\theta s - \theta e) < \pi / 2)$ Then $\theta = \theta s - 2\pi$

(B) $|\Phi(\theta s - \theta e)| > \pi / 2$

   (1) $y^* >= 0$

      (a) If $((\theta s - \theta e) > 0)$ Then $\theta = \theta s - 2\pi$

      (b) Else if $((\theta s - \theta e) < -2\pi)$ Then $\theta = \theta s - 2\pi$

   (2) $y^* < 0$

      (a) If $((\theta s - \theta e) > 2\pi)$ Then $\theta = \theta s - 2\pi$

      (b) Else if $((\theta s - \theta e) < 0)$ Then $\theta = \theta s + 2\pi$

4.   Motion Planning to Check Collision

In a motion planning simulation from a configuration $q_s = (p_s, \theta_s, k_s)$ to find a desirable path to get to the exit configuration $q_e = (p_e, \theta_e, k_e)$, it is important to check if the path collides the wall. To test it, we need to know how the world is described.

a.   *Inverted World*

We consider a two-dimensional world W with holes. A hole is an obstacle for a vehicle. A free space *Free(W)* is the complement of the union of all holes. There might be a hole among them which completely surrounds the free space. The hole like this is said to be inverted. Every free space is a connected subset of W. Figure 16 shows this world.
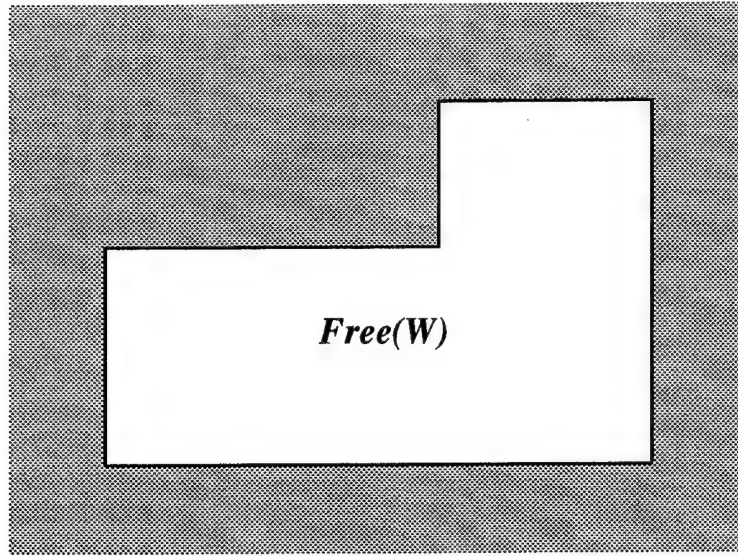
Figure 16: Inverted World

b. View Angle

With a polygon B and a point p which is not on its boundary, the view angle $\gamma_i(p)$ of the ith edge v, $\varphi(v)$ at p is defined as follows (Figure 17) :

$$\gamma_i(p) = \Phi(\Psi(p, \varphi(v_i)) - \Psi(p, v_i))$$ (Eq 6.9)
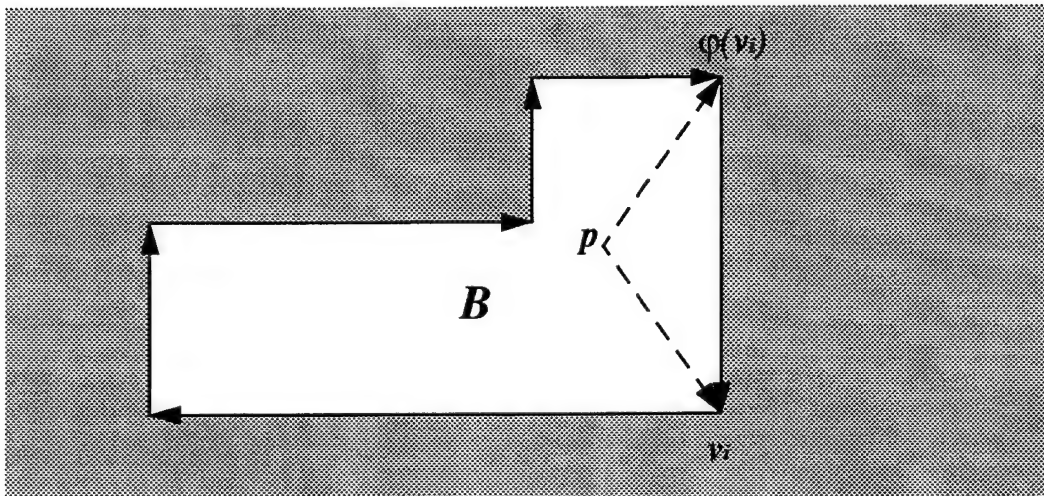
Figure 17: View Angle at point p

28

The sum of the n view angles in B is defined as

$$\Gamma_B(p) = \sum_{i=1}^{n} \gamma_i(p) \qquad \text{(Eq 6.10)}$$

In the inverted world, if $\Gamma_B(p) = -2\pi$, the point p is in *Free(W)*, otherwise, it's not in *Free(W)* [Kana].

c.   *Collision Test of Vehicle*

So far, the vehicle in the motion planning has been dealt as if one point. But it is not enough to test if the vehicle collides the walls. In real world, the vehicle has a rigid body with size such as width and length. There might be a situation while one corner of the vehicle is in the free space, the other corner could collide the wall. This behavior can not be accepted. We need to know how to test collision of each corner of the vehicle at every step of the motion planning.
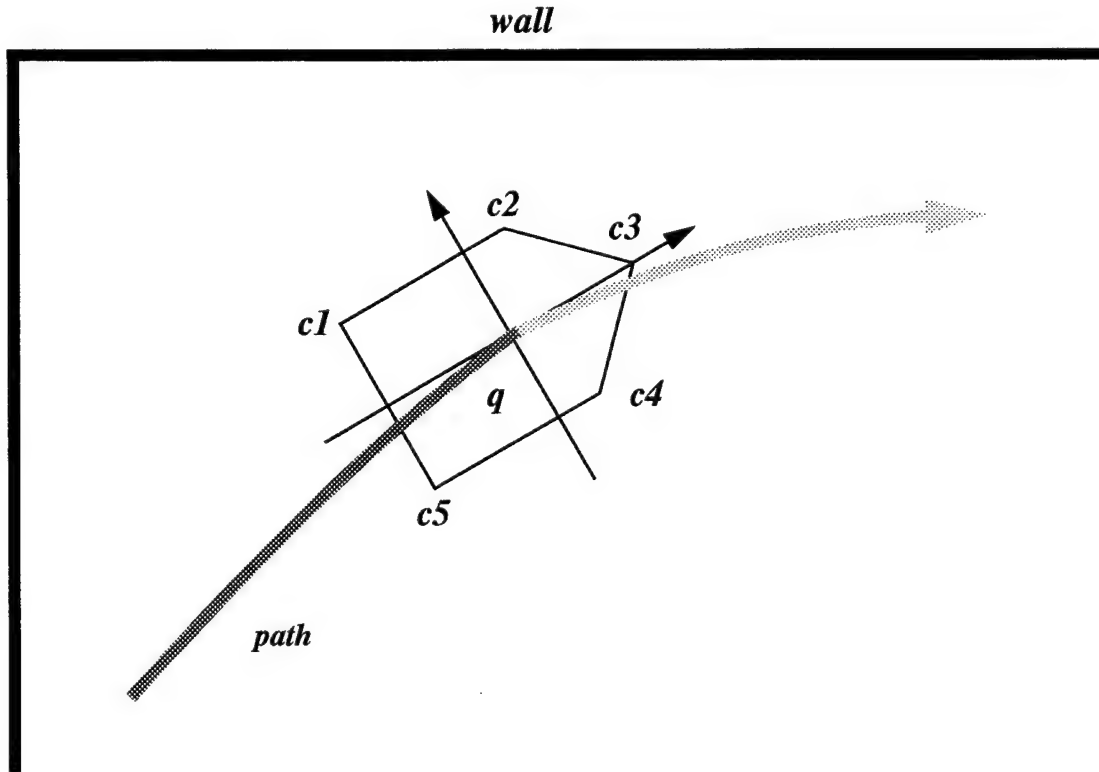
Figure 18: Collision test of vehicle

As shown in the Figure 18, we know two important informations such that the configuration of the center of the vehicle q which is on the motion planning path, and the local coordinates of vehcle's corners such as $c1, c2, c3, c4, c5$. With the information, we can get each corner's configuration as follows:

$$q1 = q \text{ o } c1,$$

$$q2 = q \text{ o } c2,$$

$$q3 = q \text{ o } c3,$$

$$q4 = q \text{ o } c4,$$

$$q5 = q \text{ o } c5$$

Instead of testing center of the vehicle, the collision test will be doing at each corner.

C. EXPERIMENTAL RESULT

Figure 19 shows the result in which the vehicle located at a configuration q_s = ((50.0, 100.0), -3π/4, 0.0) finds a path to converge to the exit configuration q_e = ((175.0, 679.704), π/2, 0.0) avoiding a collision of the walls. The default smoothness σ is 80. The vehicle used in the experiment has the size: width = 50cm, length = 60cm.
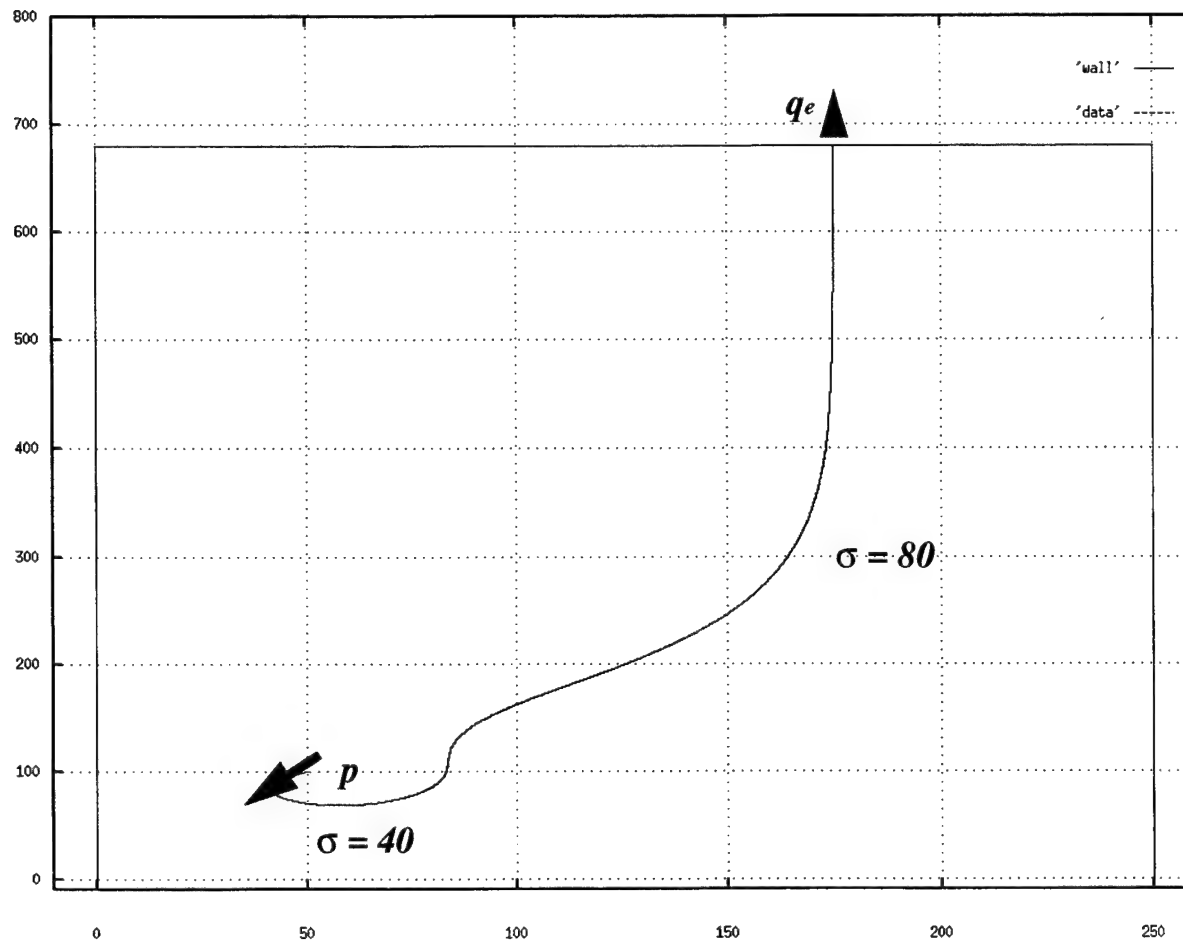
Figure 19: The result the vehicle finds a path

31

APPENDIX

A.   C PROGRAM FOR SMOOTH ROTATION

```c
/**********************************************************
  Function : rotateRule()
  Purpose  : updates the commanded velocity to rotate the robot
  Parameters: VELOCITY actual, commanded
  Returns  : The required linear velocity, rotational velocity
  Called by :
  Calls    : limit()
***********************************************************/

static  VELOCITY
rotateRule(VELOCITY actual, VELOCITY commanded)
{
  double k = 0.5;
  static double goalTheta;
  static int noGoalTheta = TRUE;
  static double rotVel = 0.0;
  double rotAcc = 0.0;

  if (noGoalTheta) {
     /* set the goal of rotation */
     goalTheta = vehicle.Theta + currentPath.config.Theta;
     noGoalTheta = FALSE;
  }

  commanded.Linear = 0.0;

  if (fabs(vehicle.Theta - goalTheta) >= 0.005) {
    rotAcc = -2 * k * rotVel - k * k *
         (vehicle.Theta - goalTheta);
    rotAcc = limit(rotAcc, DEFAULT_ROT_ACC);
    rotVel = rotVel + rotAcc * MOTION_CONTROL_CYCLE;
    rotVel = limit(rotVel, DEFAULT_GOAL_VEL_ROT);
    commanded.Rotational = rotVel;
  }
  else { /* the rotation is completed */
    currentPath.pathType.mode = STOPMODE;
    commanded.Rotational = 0.0;
    noGoalTheta = TRUE;
    rotVel = 0.0;
  }

  return commanded;
```

```
/*****************************************************
  Function  :  limit
  Purpose   :  This function is used by rotate function
               to limit rotational speed of the robot
  Parameters :  double  y, l
  Returns   :  double  y, l
  Called by :  rotateRule()
  Calls     :  None
*****************************************************/

double
limit(double y, double l)
{

  if (y > l) y = l;

  else if (y < -l) y = -l;

  return y;

}
```

B. C PROGRAM FOR SYMMETRIC MOTION PLANNING

```
/*************************************************************
  Module Name: Header file
  Purpose: define variables
  Parameters:
  Returns:
  Called by:
  Calls:
**************************************************************/

#include <iostream.h>
#include <math.h>

#define PI 3.1415926

typedef struct {
  double X;
  double Y;
}POINT;

typedef struct {
  POINT Posit;
  double Theta;
  double Kappa;
}CONFIGURATION;

CONFIGURATION symConfig[100000];

/* smoothness for virtual and real robot */
double sigma =  20;
double sigma2 = 1;

/* different delta s is used in calculatedVirtualRobotPath() */
double deltaS =  0.1;
```

```
/* constant for steering function used in simulation */
double k = 1.0 / sigma;
double a = 3 * k;
double b = 3 * k * k;
double c = k * k * k;

/* constant for steering function used in real robot */
double kk = 1.0 / sigma2;
double aa = 3 * kk;
double bb = 3 * kk * kk;
double cc = kk * kk * kk;

/* define functions used in the program */
CONFIGURATION moveRealRobot(CONFIGURATION q);
CONFIGURATION calculateVirtualRobotPath(CONFIGURATION q, CONFIGURATION L);
CONFIGURATION moveRobotOnSimulatedPath(CONFIGURATION q, int index);
double computeSymmetricPathKappa(CONFIGURATION q, int i);
double getKappa(CONFIGURATION q, CONFIGURATION L);

/* library functions */
CONFIGURATION computeQstar(CONFIGURATION q1, CONFIGURATION q2);
CONFIGURATION composite(CONFIGURATION q1, CONFIGURATION q2);
CONFIGURATION circ(double deltaS, double theta);
double norm(double angle);
```

```
/**********************************************************
  Module Name: main function
  Purpose: for symmetric motion
  Parameters:
  Returns:
  Called by:
  Calls:
***********************************************************/

main()
{
  CONFIGURATION q, L, qstar;
  static VirtualRobotMoving = 1;
  static RealRobotMoving = 0;
  static int initialSetting = 1;
  static int index = 0;
  static int i, LargestIndex = 0;

  /* robot's current configuration */
  q.Posit.X = vehicle.Posit.X, q.Posit.Y = vehicle.Posit.Y,
  q.Theta = vehicle.Theta, q.Kappa = vehicle.Kappa;

  if (initialSetting) {
    /* robot's goal configuration */
    L.Posit.X = path.Posit.X, L.Posit.Y = path.Posit.Y,
    L.Theta = path.Theta + PI, L.Kappa = -path.Kappa;
    initialSetting = 0;
  }

  /* store the goal configuration to array indexed zero */
  symConfig[index].Posit.X = L.Posit.X;
  symConfig[index].Posit.Y = L.Posit.Y;
  symConfig[index].Theta = L.Theta;
  symConfig[index].Kappa = L.Kappa;

  /* 1st part
     move real robot while simulating virtual robot */

  while (VirtualRobotMoving) {

    q = moveRealRobot(q); /* move real robot */

    /* compute new configuration path on which virtual robot moves,
       so the virtual robot L tracks the line defined by
       configuration q */
    L = calculateVirtualRobotPath(L, q);
```

37

```
          /* store new simulated configuration to array */
          index++;
          symConfig[index].Posit.X = L.Posit.X;
          symConfig[index].Posit.Y = L.Posit.Y;
          symConfig[index].Theta = L.Theta - PI;
          symConfig[index].Kappa = -L.Kappa;

          /* test xstar if the real robot passes some point
             where the real robot meets the virtual robot */

          qstar = computeQstar(symConfig[index], q);
          if (qstar.Posit.X <= 0.0) {
            VirtualRobotMoving = 0;
            RealRobotMoving = 1;
          }
        }

        /* 2nd part
           make real robot track the simulated path until the index is 0 */

        while(RealRobotMoving) {
          if (index >= 0) {

            /* test xstar if the robot passes local target configuration
               on simulated path */

            qstar = computeQstar(symConfig[index], q);

            if (qstar.Posit.X >= 0.0) {
              q = moveRobotOnSimulatedPath(q, index);
            }
            else  index--;
          }
          else {
            LEDon(4);
            currentPath.pathType.mode = STOPMODE;
            commanded.Linear = 0.0;
            commanded.Rotational = 0.0;
            initialSetting = 1;
          }
        }
      }
```

```
/***********************************************************
   Module Name: moveRealRobot function
   Purpose: move real robot step forward
   Parameters: q
   Returns: q
   Called by: main
   Calls: circ(), composite()
***********************************************************/

CONFIGURATION
moveRealRobot(CONFIGURATION q)
{
  CONFIGURATION deltaQ;
  double dtheta;

  dtheta = 0.0;
  deltaQ = circ(deltaS, dtheta);
  q = composite(q, deltaQ);
  q.kappa = 0.0;

  return q;
}
```

```
/********************************************************
  Module Name: calculateVirtualRobotPath function
  Purpose: track the virtual robot to the reference line
  Parameters: q, L
  Returns: q
  Called by: main
  Calls: getKappa(), circ(), composite()
********************************************************/

CONFIGURATION
calculateVirtualRobotPath(CONFIGURATION q, CONFIGURATION L)
{
  CONFIGURATION deltaQ;
  double dkappa;
  double theta;
  double DifferentDeltaS = 0.2;

  L.theta = L.theta + PI;

  dkappa = q.Kappa + getKappa(q, L) * DifferentDeltaS;
  theta = DifferentDeltaS * dkappa;
  deltaQ = circ(DifferentDeltaS, theta);
  q = composite(q, deltaQ);
  q.Kappa = dkappa;

  return q;
}
```

```
/*********************************************************
   Module Name: moveRobotOnsimulatedPath function
   Purpose: track the real robot on simulated path
   Parameters: q, i
   Returns: q
   Called by: main
   Calls: computeSymmetricPathKappa(), circ(0, composite()
*********************************************************/

CONFIGURATION
moveRobotOnSimulatedPath(CONFIGURATION q, int index)
{
   CONFIGURATION deltaQ;
   double dkappa, dtheta;

   dkappa = q.Kappa + computeSymmetricPathKappa(q, index) * deltaS;
   dtheta = deltaS * dkappa;
   deltaQ = circ(deltaS, dtheta);
   q = composite(q, deltaQ);
   q.kappa = dkappa;

   return q;
}
```

```
/**********************************************************
  Module Name: computeSymmetricPathKappa function
  Purpose: compute kappa for real robot on simulated path
  Parameters: q, i
  Returns: kappa
  Called by: moveRobotOnSimulatedPath()
  Calls: computeQstar()
**********************************************************/

double
computeSymmetricPathKappa(CONFIGURATION q, int i)
{
  CONFIGURATION qstar;
  double deltaKappa, deltaTheta, deltaD;

  qstar = computeQstar(q, symConfig[i]);

  deltaKappa = q.Kappa - symConfig[i].Kappa;
  deltaTheta = q.Theta - symConfig[i].Theta;
  deltaD = qstar.Posit.Y;

  return -(aa * deltaKappa + bb * norm(deltaTheta) + cc * deltaD);
}
```

```
/************************************************************
  Module Name: getKappa function
  Purpose: compute kappa for virtual robot
  Parameters: q, L
  Returns: kappa
  Called by: main
  Calls:
************************************************************/

double
getKappa(CONFIGURATION q, CONFIGURATION L)
{
  double deltaD;

  deltaD =  -(q.Posit.X - L.Posit.X) * sin(L.Theta) +
         (q.Posit.Y - L.Posit.Y) * cos(L.Theta);

  return  -(a * q.Kappa + b * (q.Theta - L.Theta) + c * deltaD);
}
```

C. C PROGRAM FOR INITIAL MOTION PLANNING

```
/********************************************************
  Module Name: Header file
  Purpose: define variables
  Parameters:
  Returns:
  Called by:
  Calls:
********************************************************/

#include <iostream.h>
#include <math.h>

#define PI 3.1415926
#define initialSigma 20
#define ABS(x) ((x < 0.0) ? -x : x)

typedef struct {
   double X;
   double Y;
}POINT;

typedef struct {
   POINT Posit;
   double Theta;
   double Kappa;
}CONFIGURATION;

/* define robot size */
POINT corner[4];
CONFIGURATION robotCorner[4];
CONFIGURATION foreRunner[1000000];

double sigma, k, a, b, c;
```

```
/* define functions used in program */
CONFIGURATION determineDirectionOfMotion(qs, qe);
void getConstantsforForerunner(double sigma);
CONFIGURATION runForerunner(CONFIGURATION q, CONFIGURATION qe);
double getKappa(CONFIGURATION q, CONFIGURATION L);
int checkIfcollide(CONFIGURATION q);
double calculateAngle(CONFIGURATION q, int i, int j);
int checkIfQparallelQE(CONFIGURATION q1, CONFIGURATION q2);
int checkIfQequalsQE(CONFIGURATION q);

/* library functions */
CONFIGURATION computeQstar(CONFIGURATION q1, CONFIGURATION q2);
CONFIGURATION composite(CONFIGURATION q1, CONFIGURATION q2);
CONFIGURATION circ(double deltaS, double theta);
double norm(double angle);
```

```
/********************************************************
 Module Name: main function
 Purpose: for initial motion
 Parameters:
 Returns:
 Called by:
 Calls:
 *********************************************************/

main()
{
    CONFIGURATION q, qs, qe, qd, qstar;
    int foreRunnerMoving = 1;
    int index = 0,  midPoint;
    int i;
    double firstSigma, dinit;
    sigma =  initialSigma;
    FILE *stream;

    /* for printing simulated path */
    stream = fopen("data", "wt");
    if(stream == NULL) fputs("File not found!\n", stderr), exit(1);

    /* robot's size */
    robotCorner[0].Posit.X = -25.0, robotCorner[0].Posit.Y =  30.0;
    robotCorner[1].Posit.X =  25.0, robotCorner[1].Posit.Y =  30.0;
    robotCorner[2].Posit.X =  25.0, robotCorner[2].Posit.Y = -30.0;
    robotCorner[3].Posit.X = -25.0, robotCorner[3].Posit.Y = -30.0;

    /* robot's initial and goal configuration */
    qs.Posit.X = 50.0, qs.Posit.Y = 100.0, qs.Theta = -3 * PI / 4.0, qs.Kappa = 0.0;
    qe.Posit.X = 200.0, qe.Posit.Y = 700.00, qe.Theta = PI / 2.0, qe.Kappa = 0.0;

    /* region where robot starts moving */
    corner[0].X = 0.0, corner[0].Y = 0.0;
    corner[1].X = 0.0, corner[1].Y = 700.00;
    corner[2].X = 200.00, corner[2].Y = 700.00;
    corner[3].X = 200.00, corner[3].Y = 0.0;

    qstar = computeQstar(qs, qe);
    dinit = qstar.Posit.Y;
    qstar.Posit.X = 0.0;

    qd = composite(qe, qstar);
    qd.Theta = qe.Theta;
```

```c
/* store the initial configuration to array indexed zero */
foreRunner[index].Posit.X = qs.Posit.X;
foreRunner[index].Posit.Y = qs.Posit.Y;
foreRunner[index].Theta = qs.Theta;
foreRunner[index].Kappa = qs.Kappa;

/* set the current configuration to start configuration */
q.Posit.X = qs.Posit.X;
q.Posit.Y = qs.Posit.Y;
q.Theta = qs.Theta;
q.Kappa = qs.Kappa;

/* set constants with sigma */
getConstantsforForerunner(sigma);

while (foreRunnerMoving) {
  q = runForerunner(q, qd);
  index++;

  /* store the new configuration to array */
  foreRunner[index].Posit.X = q.Posit.X;
  foreRunner[index].Posit.Y = q.Posit.Y;
  foreRunner[index].Theta = q.Theta;
  foreRunner[index].Kappa = q.Kappa;

  if (checkIfcollide(q) == 1) {
    sigma = sigma / 2.0;
    if (sigma < 0.095 * dinit) {
      printf("Sorry, no possible path to avoid collision!\n");
      exit(0);
    }
    index = 0;
    q.Posit.X = qs.Posit.X;
    q.Posit.Y = qs.Posit.Y;
    q.Theta = qs.Theta;
    q.Kappa = qs.Kappa;

    getConstantsforForerunner(sigma);
  }

  if (checkIfQparallelQE(q, qd)) {
    firstSigma = sigma;
    midPoint = index;
    foreRunnerMoving = 0;
  }
```

```
  } /* end of while loop */

  foreRunnerMoving = 1;
  sigma = initialSigma;

  while (foreRunnerMoving) {
    q = runForerunner(q, qe);
    index++;

    /* store the new configuration to array */
    foreRunner[index].Posit.X = q.Posit.X;
    foreRunner[index].Posit.Y = q.Posit.Y;
    foreRunner[index].Theta = q.Theta;
    foreRunner[index].Kappa = q.Kappa;

    qstar = computeQstar(qe, q);

    if (qstar.Posit.X <= 0.0) {
      if (checkIfQequalsQE(qstar))
        foreRunnerMoving = 0;
      else {
        sigma = sigma / 2.0;
        index = midPoint;
        q.Posit.X = foreRunner[index].Posit.X;
        q.Posit.Y = foreRunner[index].Posit.Y;
        q.Theta = foreRunner[index].Theta;
        q.Kappa = foreRunner[index].Kappa;

        getConstantsforForerunner(sigma);
      }
    }
  } /* end of while loop */
}
```

```
/*********************************************************
   Module Name: determineDirectionOfMotion function
   Purpose: determine the direction of robot moving
   Parameters: qs, qe
   Returns: qs
   Called by: main
   Calls: norm
*********************************************************/

CONFIGURATION
determineDirectionOfMotion(qs, qe)
{
  if (ABS(norm(qs.Theta - qe.Theta)) <= PI / 2.0) {
    if (qs.Theta - qe.Theta > PI / 2.0) qs.Theta = qs.Theta - 2 * PI;
    else if (qs.Theta - qe.Theta < - PI / 2.0) qs.Theta = qs.Theta + 2 * PI;
  }
  else if (ABS(norm(qs.Theta - qe.Theta)) > PI / 2.0) {
    if (qstar.Posit.Y >= 0.0) {
      if (qs.Theta - qe.Theta > 0.0) qs.Theta = qs.Theta - 2 * PI;
      else if (qs.Theta - qe.Theta < - 2 * PI) qs.Theta = qs.Theta + 2 * PI;
    }
    else if (qstar.Posit.Y < 0.0) {
      if (qs.Theta - qe.Theta < 0.0) qs.Theta = qs.Theta + 2 * PI;
      else if (qs.Theta - qe.Theta > 2 * PI) qs.Theta = qs.Theta - 2 * PI;
    }
  }

  return qs;
}
```

```
/*****************************************************************
  Module Name: getConstantsforForerunner function
  Purpose: compute constants a, b, c with new smoothness
  Parameters: sigma
  Returns:
  Called by: main
  Calls:
*****************************************************************/

void
getConstantsforForerunner(double sigma)
{
  /* constant for steering function used in simulation */
  k = 1.0 / sigma;
  a = 3 * k;
  b = 3 * k * k;
  c = k * k * k;
}
```

```
/***************************************************************
  Module Name: runForerunner function
  Purpose: simulate the robot with arbirtary sigma
  Parameters: q, L
  Returns: q
  Called by: main
  Calls: getKappa(), circ(), composite()
***************************************************************/

CONFIGURATION
runForerunner(CONFIGURATION q, CONFIGURATION L)
{
  CONFIGURATION deltaQ;
  double dkappa;
  double theta;
  double deltaS = 0.1;

  dkappa = q.Kappa + getKappa(q, L) * deltaS;
  theta = deltaS * dkappa;
  deltaQ = circ(deltaS, theta);
  q = composite(q, deltaQ);
  q.Kappa = dkappa;

  return q;
}
```

```
/********************************************************
  Module Name: getKappa function
  Purpose: compute kappa for virtual robot
  Parameters: q, L
  Returns: kappa
  Called by: main
  Calls:
********************************************************/

double
getKappa(CONFIGURATION q, CONFIGURATION L)
{
  double deltaD;

  deltaD =  -(q.Posit.X - L.Posit.X) * sin(L.Theta) +
        (q.Posit.Y - L.Posit.Y) * cos(L.Theta);

  return  -(a * q.Kappa + b * (q.Theta - L.Theta) + c * deltaD);
}
```

```
/***********************************************************
  Module Name: checkIfcollide function
  Purpose: test the robot's colliding of wall
  Parameters: q
  Returns:
  Called by: main
  Calls: caculateAngle()
***********************************************************/

int
checkIfcollide(CONFIGURATION q)
{
  int i, j;
  double angle = 0.0;
  CONFIGURATION qCorner[4];

  qCorner[0] = composite(q, robotCorner[0]);
  qCorner[1] = composite(q, robotCorner[1]);
  qCorner[2] = composite(q, robotCorner[2]);
  qCorner[3] = composite(q, robotCorner[3]);

  for (i = 0; i < 4; i++) {
    angle = calculateAngle(qCorner[i], 3, 0);
    for (j = 0; j < 3; j++) {
      angle = angle + calculateAngle(qCorner[i], j, j + 1);
    }

    if (angle > -PI) return 1;
  }

  return 0;
}
```

```
/*****************************************************************
   Module Name: calculateAngle function
   Purpose: calculate the view angle of robot
   Parameters: q, i, j
   Returns: angle
   Called by: checkIfcollide()
   Calls:
*****************************************************************/

double
calculateAngle(CONFIGURATION q, int i, int j)
{
  return  norm(atan2(corner[j].Y - q.Posit.Y, corner[j].X - q.Posit.X) -
          atan2(corner[i].Y - q.Posit.Y, corner[i].X - q.Posit.X));
}
```

```
/**********************************************************
   Module Name: checkIfQparallelQE function
   Purpose: test the robot is parallel to the reference line
   Parameters: q1, q2
   Returns:
   Called by: main
   Calls:
**********************************************************/

int
checkIfQparallelQE(CONFIGURATION q1, CONFIGURATION q2)
{
    double dtheta;

    dtheta = q1.Theta - q2.Theta;
    if (dtheta >= -0.1 && dtheta <= 0.1) return 1;

    return 0;
}
```

```
/*************************************************************
  Module Name: checkIfQequalsQE function
  Purpose: test the robot gets to the qe
  Parameters: q
  Returns:
  Called by: main
  Calls:
**************************************************************/

int
checkIfQequalsQE(CONFIGURATION q)
{
  if (q.Posit.Y >= -0.01 && q.Posit.Y <= 0.01) return 1;

  return 0;
}
```

D.   C LIBRARY FUNCTIONS

```
/*********************************************************
   Module Name: computeQstar function
   Purpose: calculate the local coordinate between two configurations
   Parameters: q1, q2
   Returns: qstar
   Called by:
   Calls:
*********************************************************/


CONFIGURATION
computeQstar(CONFIGURATION q1, CONFIGURATION q2)
{
   CONFIGURATION qstar;

   qstar.Posit.X = (q1.Posit.X - q2.Posit.X) *
              cos(q2.Theta) +
              (q1.Posit.Y - q2.Posit.Y) *
              sin(q2.Theta);
   qstar.Posit.Y = (q1.Posit.Y - q2.Posit.Y) *
              cos(q2.Theta) -
              (q1.Posit.X - q2.Posit.X) *
              sin(q2.Theta);
   qstar.Theta   = q1.Theta - q2.Theta;

   return qstar;
}
```

```
/*********************************************************
  Module Name: composite function
  Purpose: compose two configurations
  Parameters: q1, q2
  Returns:
  Called by:
  Calls:
*********************************************************/

CONFIGURATION
composite(CONFIGURATION q1, CONFIGURATION q2)
{
  CONFIGURATION q;

  q.Posit.X = q1.Posit.X + q2.Posit.X * cos(q1.Theta) -
          q2.Posit.Y * sin(q1.Theta);
  q.Posit.Y = q1.Posit.Y + q2.Posit.X * sin(q1.Theta) +
          q2.Posit.Y * cos(q1.Theta);
  q.Theta = q1.Theta + q2.Theta;

  return q;
}
```

```
/************************************************************
  Module Name: circ function
  Purpose: get configuration
  Parameters: deltaS, theta
  Returns: q
  Called by:
  Calls:
*************************************************************/

CONFIGURATION
circ(double deltaS, double theta)
{
  CONFIGURATION q;

  q.Posit.X = (1.0 - theta * theta / 6) * deltaS;
  q.Posit.Y = (1.0 / 2.0 - theta * theta / 24.0)* theta * deltaS;
  q.Theta = theta;

  return q;
}
```

```
/*********************************************************
   Module Name: norm function
   Purpose: normalize orientation
   Parameters: angle
   Returns:
   Called by:
   Calls:
********************************************************/

double
norm(double angle)
{
  while ((angle >= PI) || (angle < -PI))
  {
    if (angle >= PI)
      angle -= 2 * PI;
    else
      angle += 2 * PI;
  }
  return angle;
}
```

LIST OF REFERENCES

[Chuang] Chien-Liang Chuang, "Layered Motion Planning for Autonomous Vehicles", Ph.D. dissertation, Naval postgraduate School, Monterey, CA, 93943, Sep. 1995.

[Kana] Yutaka Kanayama, "Introduction to Motion Planning" Lecture notes of the Naval Postgraduate School, Department of Computer Science, Naval Postgraduate School, Monterey, CA, 93943, March 25, 1995.

[Kana95] Y. Kanayama, J Kovalchik, C. Chuang, F. Kelbe, "Motion Planning for Autonomous Mobile Robots", Proceedings of the Autonomous Vehicles in Mine Countermeasures Symposium, pp. 8.14 - 8.80, Symposium Monterey, CA, April 4 - 7. 1995

[Kova] J. Kovalchik, "Layered Motion Planning for Autonomous Mobile robots Using a Steering Function", Ph.D. dissertation, Naval Postgraduate School, Monterey CA, July 1995.

[Lato] Latombe, J., "Robot Motion Planning", Kluwer Academic Publishers, 1991.

[Loza] Lozano-Perez, T., "Spatial Planning: A Configuration Space Approach", *IEEE Transaction on Computer*, vol C-32, no. 2, pp. 108-119, Feb. 1983.

[Macp] Macpherson, D. L., "Automated Cartography by An Autonomous Mobile Robot Using Ultrasonic Range Finders", Ph.D. dissertation, Naval Postgraduate School, Monterey, CA, Sept. 1993.

[Nava] Naval Command, control and Ocean Surveillance Center, Technical Note 1194, Update 1, "Survey of Collision Avoidance and Ranging Sensors for Mobile Robots", Everett, H., and Stitz, E., Dec. 1992.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
   Cameron Station
   Alexandria, VA   22304-6145

2. Dudley Knox Library   2
   Code 013
   Naval Postgraduate School
   Monterey, CA   93943-5101

3. Chairman, Code CS   2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA   93943

4. Dr Yutaka Kanayama, Code CS/KA   2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA   93943

5. Yun, Seok Jun   2
   Seoul Dobong-Gu Dobong-Dong
   624-121   Na-Dong 101 Ho
   South Korea,   132-012

6. Woodang Library   1
   Seoul Nowon-Gu Kong Neung-Dong
   P.O. Box 77
   Korea Military Academy
   South Korea,   139-799

7. Ltc. Park, Bong Suk   1
   Chung Nam, Nonsan Kun
   Duma Myeon, Bunam Ri
   P.O. Box 4
   Dept. of Personal Education
   South Korea,   320-919